HOMEWORK 3
GRAPHICAL MODELS[1]

10-418 / 10-618 MACHINE LEARNING FOR STRUCTURED DATA (FALL 2022)
https://www.cs.cmu.edu/~mgormley/courses/10418/

OUT: Sep. 30, 2022
DUE: Oct. 10, 2022 11:59 PM
TAs: Eric, Harnoor, Mukuntha

START HERE: Instructions

Summary In this assignment, you will implement a baseline LSTM model for constituency parsing followed by a general CRF (Conditional Random Field) and train both jointly. Section 1 will help you develop a better understanding of directed and undirected graphical models through some warm-up problems. Then, in Section 2, you will build on these intuitions to implement an LSTM-CRF model and compare its performance with a vanilla LSTM.

- **Collaboration policy:** Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 2.1"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the Academic Integrity Section on the course site for more information: https://www.cs.cmu.edu/~mgormley/courses/10418/
- **Late Submission Policy:** See the late submission policy here: https://www.cs.cmu.edu/~mgormley/courses/10418/
- **Submitting your written work to Gradescope:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. Submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Each derivation/proof should be completed in the boxes provided. You are responsible for ensuring that your submission contains exactly the same number of pages and the same alignment as our PDF template. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader.
- **Submitting your Programming work to Gradescope:** You will submit your code for programming questions on the homework to Gradescope. After uploading your code, our grading scripts will autograde your assignment by running your program in a Docker container. When you are developing, check that the version number of the programming language environment (e.g. Python 3.10.4) and versions of permitted libraries (e.g. `numpy` 1.23.2 and `PyTorch` 1.12.1) match those used on Gradescope. We recommend debugging your implementation on your local machine (or the Linux servers) and making sure your code is running correctly first before submitting your code to Gradescope.

[1]Compiled on Friday 30th September, 2022 at 22:03

1 Written Questions [64 pts]

Answer the following questions in the template provided. Then upload your solutions to Gradescope. You may use LaTeX or print the template and hand-write your answers then scan it in. Failure to use the template may result in a penalty. There are 64 points and 14 questions.

1.1 Conditional Independencies

1. Consider the Bayesian Network described in Figure 1.1

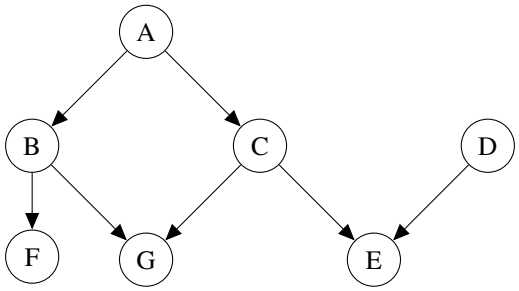


Figure 1.1: Bayesian Network Structure

Based on this network structure, answer the following questions:

(a) (1 point) Write down the equation for the joint probability distribution $P(A, B, C, D, E, F, G)$

(b) (1 point) Is $C \perp D \mid E$?

- ◯ True
- ◯ False

(c) (1 point) Is $A \perp F \mid B$?

- ◯ True
- ◯ False

(d) (1 point) Is $A \perp G \mid B$?

- ◯ True
- ◯ False

(e) (1 point) Which nodes are present in the Markov blanket of B?

(f) (1 point) Which nodes are present in the Markov blanket of D?

2. Now consider an undirected graphical model with the same set of nodes and edges as the bayesian network from figure 1.2. This model structure looks as follows:

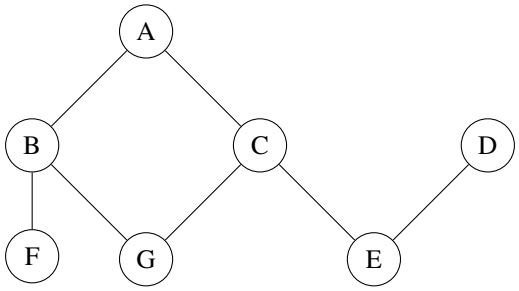


Figure 1.2: Undirected Graphical Model

For this model structure, answer the following questions:

(a) (1 point) Is $C \perp D \mid E$?

- ◯ True
- ◯ False

(b) (1 point) Is $A \perp F \mid B$?

- ◯ True
- ◯ False

(c) (1 point) Is $A \perp G \mid B$?

- ◯ True
- ◯ False

(d) (1 point) Which nodes are present in the Markov blanket of B?

(e) (1 point) Which nodes are present in the Markov blanket of D?

3. Let us now compare both models (1.1 and 1.2).

(a) (1 point) Do both models (1.1 and 1.2) have the same set of conditional independencies?

- ◯ Yes
- ◯ No

(b) (2 points) If you answered yes to the above question, list out all the conditional independencies. If you answered no, provide an example of a non-trivial graph (at least two nodes) which does have the same set of conditional independencies for both directed and undirected variants.

(c) (2 points) For the directed bayesian network, we decomposed the joint probability distribution into a product of conditional probability distributions associated with each node. However, we did not do so for the undirected model. Is it possible to write joint probability as a product of factors *without* performing marginalization (i.e. no summations) for a general undirected graphical model? Explain your answer.

1.2 Variable Elimination

1. In class, we looked at an example of variable elimination on an arbitrary graph. Let us now apply variable elimination to a familiar directed graphical model: Hidden Markov Model. A Hidden Markov Model consists of two sets of variables: X_i (observations) and Y_i (states). States are unobserved latent variables which satisfy the Markov property i.e. each state only depends on the state which immediately precedes it. Each state generates an observation. The complete structure of the model (for a sequence of length 5) looks as follows:

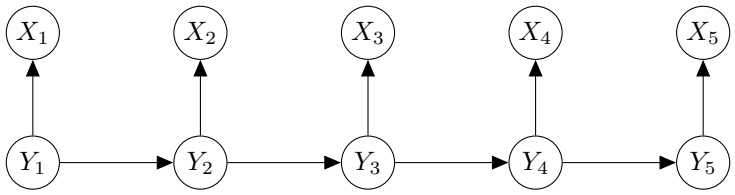
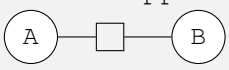


Figure 1.3: Hidden Markov Model

(a) (2 points) Draw the corresponding factor graph for this model.

```
Latex users: If you want to use tikz to draw the factor graph,
here is a sample code snippet for a tiny factor graph:
\tikz[square/.style={regular polygon,regular polygon sides=4}]
{
\node[latent] (A) {A};
\node[latent,right=1.5 cm of A] (B) {B};
\node[square, draw=black, right=0.5 cm of A] (ab) {};
\edge [-] {A} {ab};
\edge [-] {B} {ab};
}
This snippet generates the following graph:
```



(b) (2 points) For this model, write down the joint probability distribution as a product of conditional probability distributions.

(c) (4 points) Suppose we wish to compute the probability $P(Y_5 \mid X_1...X_5)$, which requires us to marginalize over $Y_1...Y_4$. Assume that we are eliminating variables in the order $Y_1 - Y_2 - Y_3 - Y_4$. Write down equations for the factors which will be computed at each step of the elimination process.

Variable Eliminated	**Factor Computed**
Y_1	
Y_2	
Y_3	
Y_4	

(d) (1 point) Is it possible to pick a better elimination order for this model?

◯ Yes

◯ No

2. In class, we saw how using variable elimination is more efficient than naively computing the joint probability. In this problem, we will further study how the order in which variable elimination is carried out affects the efficiency of this method. Consider the following undirected graphical model:

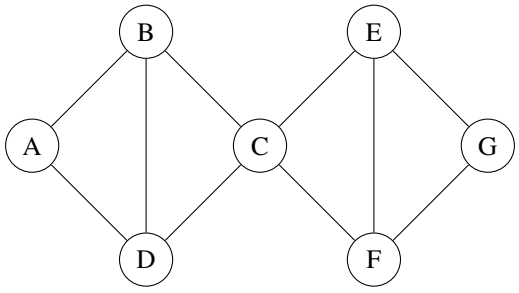
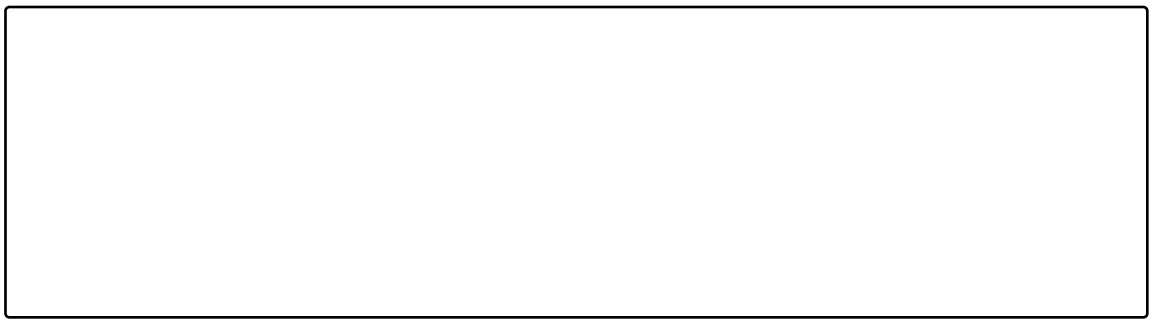


Figure 1.4: Initial graph for variable elimination

(a) (2 points) Draw a factor graph for this model, with each factor corresponding to a maximal clique in the graph



(b) (4 points) For the variable elimination order $A - G - B - D - E - F - C$, draw the intermediate factor graph at each step

Variable Eliminated	**Intermediate Factor Graph**
A	
G	
B	
D	
E	
F	
C	

(c) (4 points) For the variable elimination order $C - B - E - A - D - F - G$, draw the intermediate factor graph at each step

Variable Eliminated	**Intermediate Factor Graph**
C	
B	
E	
A	
D	
F	
G	

(d) (1 point) Which of the above elimination orders is better and why?

(e) (1 point) Based on your observations, can you think of a way to estimate which elimination order is better without going through the complete process?

1.3 Message Passing

a	$\psi_A(a)$
0	1
1	2

b	$\psi_B(b)$
0	2
1	1

c	$\psi_C(c)$
0	1
1	1

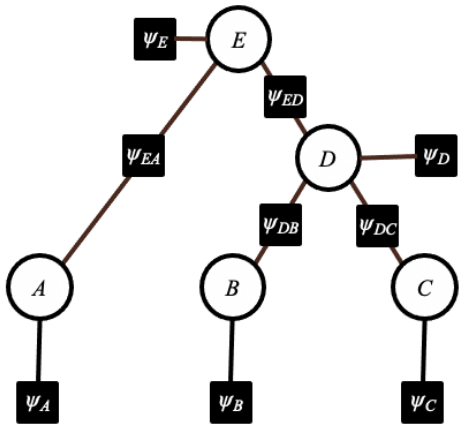


Figure 1.5

d	$\psi_D(d)$
0	1
1	1

e	$\psi_E(e)$
0	1
1	2

a	e	$\psi_{EA}(a,e)$
0	0	1
0	1	1
1	0	1
1	1	1

d	e	$\psi_{ED}(d,e)$
0	0	1
0	1	1
1	0	2
1	1	1

b	d	$\psi_{DB}(b,d)$
0	0	1
0	1	2
1	0	1
1	1	1

c	d	$\psi_{DC}(c,d)$
0	0	1
0	1	1
1	0	1
1	1	3

Consider the factor graph in Figure 1.5. On paper, carry out a run of belief propagation by sending messages first from the leaves ψ_A, ψ_B, ψ_C to the root ψ_E, and then from the root back to the leaves. Then answer the questions below. Assume all messages are un-normalized.

1. (1 point) **Numerical answer:** What is the message from A to ψ_{EA}?

2. (1 point) **Numerical answer:** What is the message from ψ_{DB} to B?

3. (1 point) **Numerical answer:** What is the belief at variable A?

4. (1 point) **Numerical answer:** What is the belief at variable B?

5. (1 point) **Numerical answer:** What is the belief at factor ψ_{DB}?

6. (1 point) **Numerical answer:** What is the value of the partition function?

1.4 Empirical Questions

The following questions should be completed after you work through the programming portion of this assignment (Section 2).

1. (1 point) **Select one:** If you feed the inputs shown in Figure 1.5 into your belief propagation module implemented in PyTorch do you get the same answers that you worked out on paper? *(Hint: The correct answer is "Yes".)*

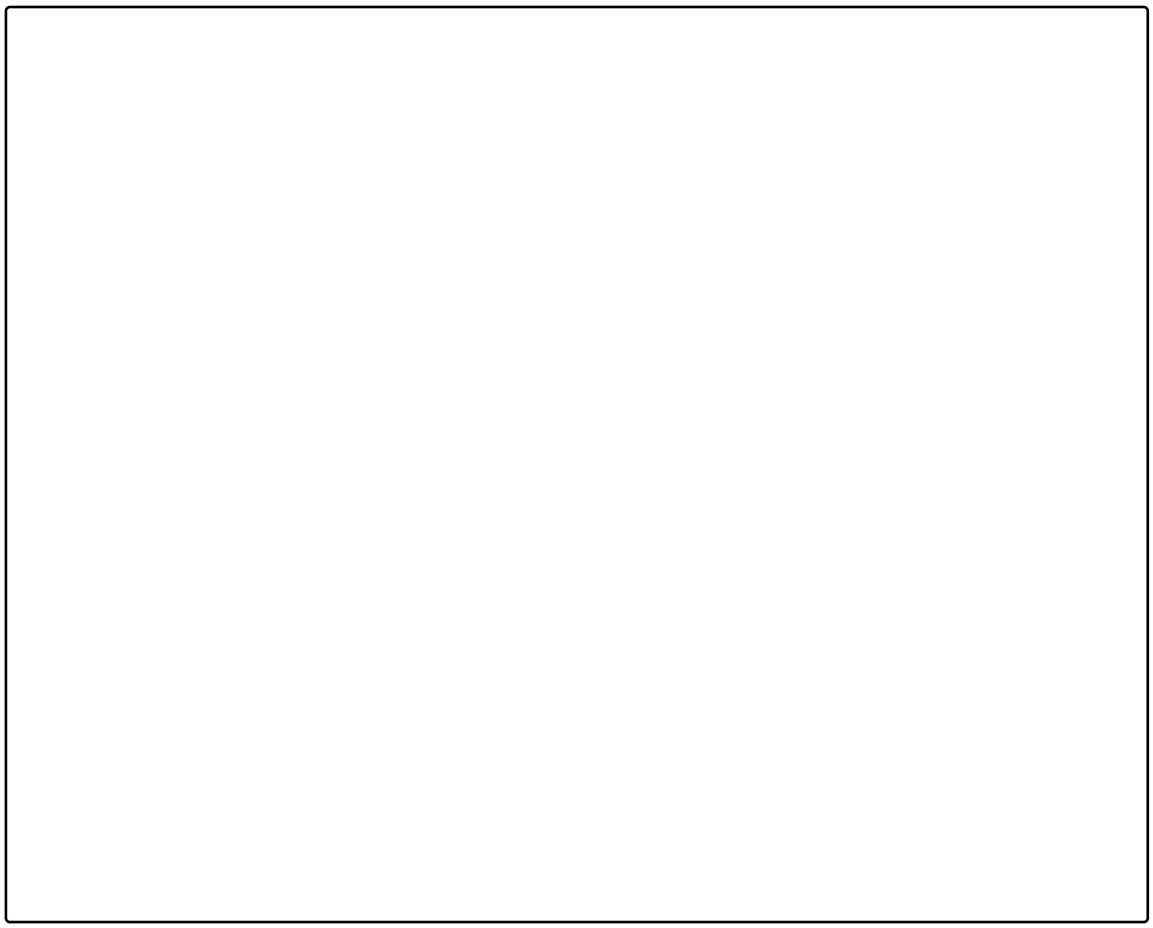
   ◯ Yes

   ◯ No

2. (10 points) Record your model's performance after three epochs on the test set and train set in terms of Cross Entropy (CE) , accuracy (AC) and leaf accuracy (LAC). *Note: Round each numerical value to two significant figures.*

Schedule	**Baseline**	**CRF**
Training CE		
Training AC		
Training LAC		
Test AC		
Test LAC		

3. (10 points) Plot training and testing cross entropy curves for : *Baseline*, *CRF Model*. Let the x-axis ranges over 3 epochs. *Note: Your plot must be machine generated.*



1.5 Collaboration Policy

After you have completed all other components of this assignment, report your answers to the collaboration policy questions detailed in the Academic Integrity Policies for this course.

1. Did you receive any help whatsoever from anyone in solving this assignment? If so, include full details including names of people who helped you and the exact nature of help you received.

2. Did you give any help whatsoever to anyone in solving this assignment? If so, include full details including names of people you helped and the exact nature of help you offered.

3. Did you find or come across code that implements any part of this assignment? If so, include full details including the source of the code and how you used it in the assignment.

2 Programming [35 pts]

Your goal in this assignment is to implement a CRF belief propagation algorithm for constituency parsing. Given the structure of the tree, you will implement a model to label the nodes with the appropriate tag. Your solution must be implemented in **PyTorch** using the data files we have provided. We have also provided template code for you to use.

2.1 Background: The Constituency Tree Labeling Task

Constituency parsing aims to extract a parse tree from a sentence that represents its syntactic structure according to a phrase structure grammar. Terminals are the words in the sentence, non-terminals in the tree are types of phrases, and the edges are unlabeled. The pre-terminals (i.e. nodes just above the leaves, aka words) are called part-of-speech tags. The other non-terminals are clause or phrase level tags. For more information on tags, look here. Throughout this assignment, we use *nodes* to refer to the set of all non-terminals.

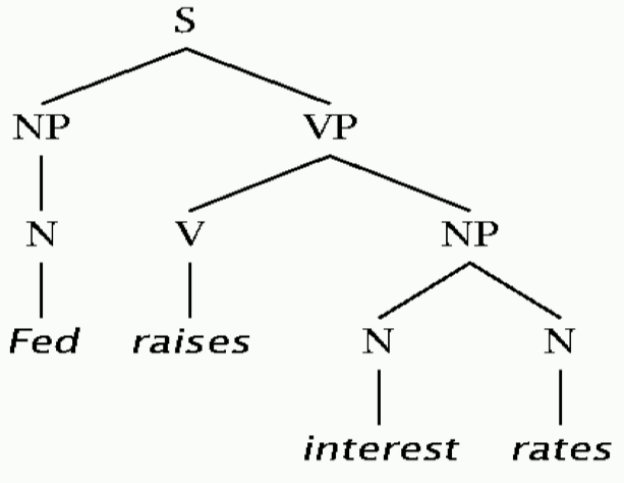


Figure 2.1: A parse tree with eight non-terminals: four part-of-speech tags (N, V, N, N) and four phrase-level tags (NP, VP, NP, S).

In this assignment, we will assume that for each example, the *branching structure* of the tree is *known*, but the tags are not. Given the structure, our goal is to successfully predict the appropriate tag for each node in the tree. We define the accuracy of the model as the average accuracy over all examples where each example consists of a tree structure with T nodes and L leaf nodes. Accuracy for a single tree is defined as:

$$\text{Acc} = \frac{\text{number of correctly predicted nodes}}{T}$$

Note that this accuracy is computed across all nodes in the graph. In practice, however, we may particularly care about the POS tags corresponding to each word in the sentence. Thus, we define leaf accuracy as:

$$\text{Leaf Acc} = \frac{\text{number of correctly predicted leaf nodes}}{L}$$

2.2 Background: The Data

We have provided a pre-processed version of Penn Tree Bank with 13,000 examples, divided into 10,000 training examples `ptb-train-10000.txt`, 1,000 development examples `ptb-dev-1000.txt`, and 2,000 test examples `ptb-test-2000.txt`. Each line consists of one tree. For example, the tree shown above would be encoded as:

```
(S (NP (N Fed)) (VP (V raises) (NP (N interest) (N rates))))
```

The trees have already been binarized such that each node has at most two children. We have provided **starter code** to read each line into an NLTK tree data structure, and a custom tree structure, which you can modify. Note that some functions below have already been implemented for you, however it is your responsibility to read through the code carefully and understand how each function works, because they will be important for later functions that you will have to implement.

The task has the following input/output:

- *Given Input:* An input sentence and the associated skeleton of its constituency parse tree
- *Predicted Output:* The labels of the non terminals in the parse tree

2.3 Baseline Model: LSTM with independent tag predictions

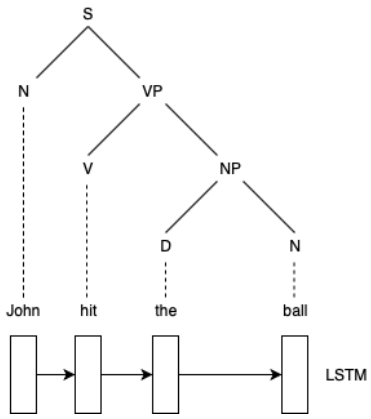


Figure 2.2: Baseline model using a unidirectional LSTM.

In this section, we have implemented a working baseline LSTM model. Starter code for the model can be found in `baseline_template.py`. As previously mentioned, you are responsible for reading through and understanding this code, because it will help make implementation of the main model easier. Below is an outline/description of the starter code given.

First, we use a torch.nn.Embedding layer to convert our sentence to a tensor representation. Then, we use an LSTM layer (torch.nn.LSTM) to output a "hidden state" for each node. For the leaf nodes (i.e. nodes corresponding to each token in the sentence), this hidden state is simply the corresponding output of the LSTM. For intermediate nodes, we use a linear layer on the concatenation of the hidden states of the left and the right child. If there is only one child, we concatenate its hidden state with itself. We use the direct child nodes, *not* the leftmost descendent and rightmost descendent. The output should be a distributions over tags. We can then train the model using cross entropy loss.

Our implementation has a single-layer unidirectional LSTM which outputs a hidden state of dimension 128. The embedding size should be 256. The optimizer should be set to be Adam with a learning rate of 0.0001. Due to the complexity associated with building the tree and computing its potentials, we use a batch size of 1.

2.4 Main Model: LSTM + CRF

In this section, you will implement a CRF layer on top of an LSTM representation (Figure 2.2). This will involve computing the unary potentials (a.k.a. factors) corresponding to each node and binary potentials corresponding to each edge in the tree. For an example of unary and binary potentials, see problem A.3. Starter code can be found in `lstm_crf_template.py`. Below is an outline of the model, note that to make the assignment more manageable we have given parts of the implementation to you in the starter code (again it is your responsibility to read through these implementations to understand what the code is doing and for easier implementation of the TODOs). All programming portions that you are required to carry out are marked with TODOs in the starter code.

1. **Representation of potentials.** Instead of keeping an explicit lookup table for the unary and binary potentials, these potentials will be computed by applying a linear layer to the LSTM hidden representation computed for each node (or in the case of binary potentials, concatenation of hidden representations). We compute a unary potential for every node in the graph and an edge potential for every edge. Note that the dotted lines in Figure 2.2 do not count as edges.

   To ensure that the potentials are positive and to provide better numerical stability, we assume that the output of the linear layer is the *logarithm* of the potentials. From there, **all further computation should be carried out in logspace**.

2. **Belief propagation.** Now that you've set up the unary and binary potentials, it's time to implement belief propagation. The sum-product belief propagation algorithm proceeds as follows:

   (a) *Send messages from the leaves to the root.*

   As our implementation is a pairwise MRF, we can send messages from node to node directly. Hence, we don't need to calculate intermediate factor node messages. We compute each message as:

   $$m_{i\to j}(y_j) = \sum_{y_i} \phi_i(y_i)\phi_{ij}(y_i, y_j) \prod_{k\in N(i)\setminus j} m_{k\to i}(y_i)$$

   for an acyclic undirected graphical model defined by $G = (V, E)$ where each element $i \in V$ of the vertex set is an index with a corresponding variable y_i for that node, and $\mathcal{N}(i)$ are the neighbors of a node i. Messages sent from one variable y_i to another variable y_j is denoted as $m_{i\to j}(y_j)$.

   (b) *Send messages from the root to the leaves*: After having computed all messages to send upwards to the root, we can re-use some of these messages when sending messages downward from the root.

   (c) *Compute beliefs*: Given these messages, we are able to compute the beliefs and hence the marginals at each node. Variable beliefs can be computed as:

   $$b_i(y_i) = \phi_i(y_i) \prod_{j\in N(i)} m_{j\to i}(y_i)$$

. Then variable marginals can be computed as:

$$p(y_i = k) = \frac{b_i(k)}{\sum_{l \in \mathcal{Y}} b_i(l)}$$

3. **Useful trick: logsumexp** When numbers are too small or too large, precision can be lost to floating point errors and overflow errors can occur. To circumvent this numerical stability issue, we can compute all messages and potentials in the log space. Multiplication operations on messages would then become addition of log messages, since $\log(ab) = \log(a) + \log(b)$. Addition of messages can be done using the logsumexp operation. Available in pytorch as torch.logsumexp, this function is equivalent to exponentiating the elements of a tensor, summing them along a specified dimension, and then taking the log to put everything back in logspace.
4. **Loss function: negative log-likelihood.** The loss function should be the negative log-likelihood (NLL), computed from the CRF potentials. This can be done for each sample by looking up and adding up the potentials associated with each node's true label, and subtracting out the partition function. (Hint: How do you compute the partition function from your log belief? Can you think of any sanity checks that can come from the computing the partition function?)
5. **Computing accuracy.** To help evaluate your model, compute the accuracy for each tree as well as the accuracy among the leaves. To do so, predict the tag with highest marginal probability for each node, and compare against the true labels.

Additional details: Again, use a hidden dimension of 128 and embedding dimension of 256. Use the Adam optimizer with learning rate 0.0001.

2.5 Code Submission [35 pts]

You must submit all of your code to the appropriate slot on Gradescope.